Improving malware detection response time with behavior-based statistical analysis techniques

Dumitru Bogdan Prelipcean
"Al. I. Cuza" University - Faculty of Computer Science
Bitdefender Laboratory
Iaşi, România
Email: bprelipcean@bitdefender.com

Adrian Ştefan Popescu
"Al.I. Cuza" University - Faculty of Computer Science
Bitdefender Laboratory
Iaşi, România
Email: apopescu@bitdefender.com

Dragoş Teodor Gavriluţ
"Al. I. Cuza" University - Faculty of Computer Science
Bitdefender Laboratory
Iaşi, România
Email: dgavrilut@bitdefender.com

***Abstract*—Detection of malicious software is a current problem which can be solved via several approaches. Among these are signature based detection, heuristic detection and behavioral analysis. In the last year the number of malicious files has increased exponentially. At the same time, automated obfuscation methods (used to generate malicious files with similar behavior but different aspect) have grown significantly. In response to these new obfuscation methods, many security vendors have introduced file reputation techniques to quickly find out potentially clean and malicious samples. In this paper we present a statistical based method that can be used to identify a specific dynamic behavior of a program. The main idea behind this solution is to analyze the execution flow of every file and to extract sequences of native system functions with a potential malign outcome. This technique is reliable against most forms of malware polymorphism and is intended to work as a filtering system for different automated detection systems. We use a database consisting of approximately 50.000 malicious files gathered over the last three months and almost 3.000.000 clean files collected for a period of 3 years. Our technique proved to be an effective filtering method and helped us improve our detection response time against the most prevalent malware families discovered in the last year.**

Keywords: malware detection, statistical analysis, malicious behavior,large databases

I. Introduction

Considering the exponential growth of malicious files number[1], it is critical to develop new ways to filter and detect malicious content, ways that are also feasible in terms of computational speed and memory use. The classical detection methods based on signatures, heuristics and runtime monitoring can be improved by using automated systems for extracting signatures, features and behavior characteristics in order to keep up-to-date with new threats. Furthermore, nowadays malware is designed to avoid detection - malicious files are specially crafted so that security products do not identify them as a threat at the moment they are released. This means that security products must also develop a secondary classification system that is used internally and has a completely different approach than the detection methods used on end user systems. To develop a method that is used to identify malicious content, one must take into consideration several factors like detection rate, memory footprint, computer slowdown, etc. Building a method that is used internally is free of some of the limitations previously described. Memory footprint and computer slowdown are not a concern as more powerful hardware can be used to overcome these problems. Also detection rate is not the main focus as these methods are designed to work as complementary to other filtering and detection systems. Last but not least, the usage of virtual environments allows us to fully analyze the behavior of every malicious file and extract valuable information that can be used by different classification algorithms.

This paper presents a system that uses data that can be extracted from virtual environments in order to create a method to filtering malicious content. By using the behavior information that was identified for each malicious file we were able to create a classifier that is not affected by most obfuscation methods. This also allowed us to track down multiple changes that malware families experience in order to avoid detection.

The paper is structured into the following sections: section II presents the technical advances in this field of research, section III focuses on describing the theoretical approach that we chose to use, section IV presents the results that we have obtained on large databases and, finally, section V presents the conclusion and future work.

[1] http://www.av-test.org/en/statistics/malware/

978-1-5090-0461-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SYNASC.2015.44

II. Related work

There are not many studies on the filtering problem and most of them focus on malware filtering over computer networks based on package analysis and network load[4][3]. As our approach focuses on application behavior and statistical analysis, we will synthesize the results on the detection problem.

One approach for behavior detection uses model-checking techniques to specify a malicious behavior[2][7]. In this case, the behavior is specified by a CTL/LTL[2] based formula. For this formula, detection boils down to verifying formula against program model by using classical model-checking algorithms. Writing this kind of specifications for one behavior requires human-driven analysis. This solution can cover the behavior of a malware family and it is false positive proof. But as the number of malware samples and families grows the solution increases the costs and has a long response time. Automatic behavior extraction is a helping stage on writing behavior specification. This can be accomplish by comparing the behavioral properties of benign files with those of malicious or suspicious files. In Christodorescu et. al[5], dependency graphs on executed code were used as behavioral property. The differences between dependence graphs extracted for benign and malicious instances represent malware specifications. An automated extraction method for a model-checking approach[13] was tested in 2013 by mining specifications through static reachability analysis using system call dependency trees[9]. These solutions are based on the control flow of an application, which is the sequence of instructions executed or actions performed by the application. A different solution based not on control flow but on data flow, is described in [8]. The assumption behind this solution is that the different variants of a specific malicious application, have at some point the same data loaded in memory. The disadvantage is the large amount of data that has to be analyzed. So, for a large number of samples a feasibility issue may occur when applied in a real life scenario.

These methods focus on deterministic modeling and extraction of a behavior or structural properties. Applying them to a large number of malicious applications is difficult. The behavioral diversity also increases the complexity of analysis procedures. In this context, the use of statistics-based or probabilistic methods is necessary. One such approach is focused on byte sequences randomness testing[10]. According to this method the sequences of bytes that are not likely to appear in a binary file are considered a malicious property. In [14], a byte-level statistical analysis using n-grams is described. Because of the low-level information that is used, these methods are not resistant to malware obfuscation and packing. Another approach that surpasses this problem by translating the byte-level into a higher level set is presented in [12]. The drawback is the used ratio between clean and malicious samples that can cause false positives on real life scenarios. In [1], the extracted information is the flow of native system calls. Even though the results were good, the experiments used only 314 samples divided in 72 benign files and 242 malicious files. This small order of magnitude and ratio is likely to cause totally different results when tested in real world conditions. An approach that uses frequency analysis of the native system calls is described in [6]. This method has good results on identifying similar behaviors. But due to the fact that it only considers the statistic distribution and not the flow of the native system calls, collisions and false positives may appear.

Our approach uses, as behavioral feature, sequences of native system calls and its purpose is to find those subsequences that are likely to describe a malicious behavior. The features identified are used afterwards as signatures for a filtering method. This filtering method is used as a preprocessing stage of an incoming set of files for an analysis system.

[2]Computational Tree Logic/Linear Temporal Logic

III. Description of the problem and the algorithms involved

A. Current malware landscape

To understand better the problem we are dealing with, we have to describe first what is the current malware landscape. In the early days of developing software with malign intentions, the authors had a strong desire for recognition. Therefore the malicious code was very easy to analyze and identifying variations of an already known malicious code was almost trivial. The purpose of these malware was usually to spread themselves to as much systems as they could. In the following years, computers started to be used not only in institutions but also with a personal purpose. This is the time when malware became indeed ”malicious”, destructive and financially orientated. Its purpose ranged from damaging personal systems, stealing authentication details or even creating a channel to control a system from a remote location without any knowledge of its owner.

At the same time with the evolution of malware people began to realize the necessity of a security solution in order to stop the execution on theirs systems of malicious applications. As security solutions started to be used on a global scale, the malware authors faced the following problem: how to create a malware that will evade detection in order to infect a system. The evasion techniques can vary from adding useless or scrambled code to more complex methods, in order to hide the malicious content from automated analysis systems. One such technique is to use a polymorphic engine[15] in order to generate new and much more code (useless or not), in order to perform the

same functionality. This technique is preferred by malware authors because it guarantees that the new code is fulfilling the same scope as the original one.

Modern operating systems are modeled in such a way that, in order for an application to accomplish any action, it will need to communicate with it through APIs [3]. APIs are specific functions and protocols that are used to describe a certain functionality. These functionalities can vary from accessing a file or handling processes to capture keystrokes or opening a channel of communication to another system. On the operating systems, APIs are usually grouped by their purpose and relevance in libraries. Thus, any application that runs inside the operating system can access these libraries and the functions (APIs) that they define. During the execution of an application multiple libraries can be loaded in order to be able to call APIs in a specific sequence for more complex program functionality.

Taking into consideration that malwares usually do not have only simple functionalities (like opening and closing a file), but multiple and more complex ones (such as opening a file, generating an encryption key and encrypting the opened file, blocking the system in order to recover it and so on), we can state that a malware has its own specific behavior. Due to the fact that any communication with the operating system for performing an action must be done using system APIs, we can also state that the APIs that are used and the sequence in which they are called by the application can outline its behavior.

The tendency of using polymorphic engines by malware authors is a big issues for security vendors due to the fact that the same code can be used as input for multiple such engines. Thus, starting from one file and using multiple polymorphic engines a malware author can generate thousands or even millions of similar files with the same behavior. Each polymorphic technique can have a different algorithm to generate code in order to better evade the detection. Usually, the particularity of an application that is not changed is its behavior. This is the feature that is rarely changed, not only from one file to another, but also on different versions of the same malware. In order to better detect different versions of the same malware family, a security solution would try to identify both the specific behavior for that family and its malicious code.

Nowadays, the security vendors are also confronted with another problem. The malware authors are using online services (either publicly available[4] or private), based on latest detections of security vendors in order to check if files generated using a polymorphic engine will be detected if deployed on the Internet. This knowledge around one or more polymorphic engines will help evading anti-virus detection. Suppose, for example, that a malware targets Internet banking users in Brazil. That malware author will change the polymorphic technique until the file generated is no longer detected by security vendors in Brazil. Only after the results are satisfying, the malware will be deployed to the targeted systems. Also, once the file is detected, the technique will be improved or even changed in order to further evade the detection and the entire process restarts. In these cases, it is obviously very important for a security vendor to deploy a generic detection. However, another complementary method, unavailable for testing to the malware authors should be available so that a security vendor is prepared for the next loop of evasion techniques.

Therefore, in order to create an up to date detection with a reasonable small response time, a security vendor will need not only a good detection technique, but also a different filtering method available for internal usage. This way, a malware writer will not be able to target this method using polymorphic code.

B. Problem description

Taking into consideration the malware landscape previously described, the growing number of malicious files and the high number of files with a suspicious behavior, we can be certain that some security solutions have automated systems that add detection in order to identify them as fast as possible. Apart from these systems that are generating detection for malicious files, a complementary heuristic detection for larger malware families is usually available. If we can assume that most of the heuristic detection is focused on prevalent malware families or on files with a certain set of suspicious features, then we can also assume that the automated systems are focused on all variety of malware or suspicious files.

Thus, we are wondering how we can prioritize the files which represent the input for automated detection systems, in order to have a better response time for a new undetected malicious file. At the same time, a method for identifying new files possibly belonging to a well known family is also needed, in order to improve generic detection. Both requests should be fulfilled by a internal filtering system and not be a detection method that is available for testing with a deployed product.

The system is intended to create a specific order in which a set of files will be analyzed and at the same time will generate more meta-data in order to improve generic detection. This type of system should not be publicly available. Thus, any new intelligence generated by this system will be available only with the purpose of improve detection and the response time for undetected malicious samples.

C. Formal model and analysis

We will describe in this section the formal model of our problem and the statistical analysis method.

[3] Application Programming Interface

[4] https://www.virustotal.com, http://virusscan.jotti.org/en, etc

1) Trace: The concept of *trace* is one which will be intensively used in our paper. By *trace* of a program we may understand:

1) A sequence of binary executed instructions, obtained by executing a file on an controlled and monitored environment.
2) A sequence of effects which are obtained by executing the program application in a controlled environment and observing the differences against the initial state of the environment.
3) A sequence of high-level function calls, obtained by executing a file on a controlled and monitored environment.

The atomic symbols of a trace, named *tokens*, are elements of a finite set V. These elements can be considered as raw data or in an idealized form. In the case of idealized form, the relevant and simplified information will be extracted using an algorithm, thus adding more computational time to the extraction process. A *trace* ω over V is a finite sequence of tokens that is a word over V. The $|\omega|$ is the length of trace ω. We present in Table I a trace obtained from binary executed instructions and differences between tokens in raw data and in the idealized form. The idealized form of tokens replaces the constant arguments with the argument type (C is used for a constant and M for a memory address).

Raw tokens	Idealized form
push eax	push eax
mov eax,0x20	mov eax, C
mov [0x42565A],eax	mov M, eax
pop eax	pop eax

Table I
EXAMPLE OF A TRACE FROM BINARY EXECUTED INSTRUCTIONS

2) Contextual trace: Our purpose is to describe behaviors and to identify those that are considered suspicious or malicious. A single token cannot define a behavior. The information from a single token does not offer any clue about a behavior, but a sequence of tokens increases the expressive power. A trace of a program can contain subsequences of tokens that are related to a clean behavior and, in the similar way, subsequences of tokens that are related to a malicious behavior. In order to identify this subsequences we use a structure inspired by *n-grams* used in computational linguistics, which, in this case, are built over the set of tokens instead of words or strings. We are using n-gram models in order to strictly define a context for a given token. Further on, we introduce the notion of *contextual trace*, also named in this paper as *context*, were there are no ambiguities.

Let P be the model of an executable program and $\omega_P = x_1, ..., x_n$ be a trace of P, where $x_i \in V, \forall 1 \leq i \leq n$. A contextual trace (or context) of x_i in ω_P is 3-tuple $(pre, x_i, post)$ where $pre = x_m...x_{i-1}$,$post = x_{i+1}...x_k$ with $m \leq i-1$, $k \geq i+1$. The following notations will be used further:

- $C(x_i, \omega_P)$ - denotes the context of x_i token in trace ω_P. It is equivalent with the 3-tuple described above.
- $\bullet C(x_i, \omega_P)$ - denotes the left context of x_i token in trace ω_P, equivalent with pre sequence from the 3-tuple.
- $C \bullet (x_i, \omega_P)$ - denotes the right context of x_i token in trace ω_P, equivalent with $post$ sequence from the 3-tuple.
- $C(\omega_P)$ - denotes the set of all contexts for trace ω_P.

These contextual traces have the same order as tokens in the trace they are built from. A sequence of contexts is one or more consecutive contexts. Two consecutive contexts have the central token from the 3-uple consecutive in the trace they are built form. Considering this we can expand a sequence of contexts to the corresponding sequence of tokens.

3) Statistical analysis: We study the population of extracted traces from a set of programs. Considering the problem of malware detection, this population has to be divided in two classes: malicious and clean (benign). We will consider two sets, one for traces extracted from clean programs $\mathcal{B}$ and one for traces extracted from malicious programs $\mathcal{M}$. Because different programs with different aspects may have the same extracted traces, we will also define multisets for the clean and malicious traces as follows:
$TrB : \mathcal{B} \to \mathbb{N}$ respectively $TrM : \mathcal{M} \to \mathbb{N}$. A trace $\omega \in \mathcal{B}$ or a trace $\omega' \in \mathcal{M}$ is in the corresponding multiset if and only if $TrB(\omega) > 0$, respectively $TrM(\omega') > 0$. We know that the set of tokens V and multiset of traces TrB are finite. Also the length of a trace and length of a context are bounded.

Therefore, for each $x_i \in V$ exists $\omega_j, \omega_k \in TrB, \omega_j \neq \omega_k$ such that $C(x_i, \omega_j) = C(x_i, \omega_k) = (pre, x_i, post)$. In this case, we will also consider the multiset of contextual traces for the multiset of clean traces TrB as $CtxB$, defined as follows:
$CtxB : Contexts \to \mathbb{N}$, where $Contexts$ is the underlying set of the multiset. An element $c \in Contexts$ is in the multiset $CtxB$ if and only if $CtxB(c) > 0$. The cardinal of multiset $CtxB$ is the sum of occurrences of elements of the underlying set.

$$|CtxB| = \sum_{i=0}^{|Contexts|} CtxB(c_i) \text{ with } c_i \in Contexts$$

For every $\omega \in TrB$ and for every $x_i \in \omega, 1 \leq i \leq |\omega|$ we build the context $C(x_i, \omega)$ and we add it in the multiset $CtxB$. In the end, we will have the set of all contexts that appear in the clean traces and the multiplicity function of the multiset gives us the number of occurrences.

Let A_c be the event that a context $c \in Contexts$ occurs

in the multiset of clean traces TrB. We can compute the probability of event A_c:

$$P(A_c) = \frac{CtxB(c)}{|CtxB|}$$

We use this probability to find the contexts that are not likely to occur in clean traces. This means that a sequence of contexts with low probability is likely to describe a malicious behavior. In order to find the mentioned sequences of contexts we will have to establish a threshold for the probability $P(A_c)$. We will denote the threshold as th. This threshold is the minimum from all occurrence probabilities of contextual traces from the multiset of clean traces.

$$th = min(P(A_c)),\ \forall c \in Contexts, CtxB(c) > 0$$

We further describe a sample analysis method. For a given file F we build the trace ω_F. For every token $x_j \in \omega_F$ we build the context $C(x_j, \omega_F)$. If this context exists in the multiset $CtxB$ and its probability of occurrence is less than the threshold th then it is likely that this context belongs to a description of a malicious behavior. If it does not exist in the multiset $CtxB$ we consider that the context is suspicious. Further we add this context to the subsequence of suspicious contexts. We will call this subsequence of suspicious contexts a $zone$. Every such $zone$ contains only consecutive contexts. The sequence of elements $zone$ is considered to describe an unknown or malicious behavior in F. We denote this sequence Z_F where each zone is delimited by a special element ($\notin V$), $newZoneMark$. If no $zone$ is found, the trace is considered to be similar with a clean one. This method is described in Algorithm 1.

We extract the suspicious zones for every instance M_i from the set of malicious instances $\mathcal{M}$.

We use a labeling function, $Label$, which for a given zone, returns a sequence of fixed length labels. A label is computed as a MD5[11] hash on the expanded contexts from a zone, which is a sequence of tokens.

For the following example of zones $(c_1, c_2, c_3, newZoneMark, c_7, c_8)$ the $Label$ function will return $label_1 label_2$, where $label_1$ is the corresponding label for the subsequence c_1, c_2, c_3 and $label_2$ for c_7, c_8.

This sequence of labels is used as a signature for the malicious behavior. These signatures are stored and then used to identify samples that have the same malicious properties.

Algorithm 1 Extraction method for sequences of zones

```
1: function ZONES(Ctx, F, th)
2:     Z_i ← ∅
3:     idx ← 0
4:     ω_F ← getTrace(F)
5:     for each x_j ∈ ω_F do
6:         c ← C(x_j, ω_F)
7:         if Ctx(c) > 0 then
8:             p ← P(A_c)
9:             if p < th then
10:                if |idx − j| > 1 then
11:                    Z_i.add(newZoneMarker)
12:                end if
13:                Z_i.add(c)
14:                idx ← j
15:            end if
16:        else
17:            if |idx − j| > 1 then
18:                Z_i.add(newZoneMarker)
19:            end if
20:            Z_i.add(c)
21:            idx ← j
22:        end if
23:    end for
24:    return Z_i
25: end function
```

D. Practical considerations

1) Statistical observations and analysis methods: In our experiments, we used as tokens the native API calls. The traces were extracted using a proprietary internal tool from Bitdefender Laboratories. The output has the following information: the name of the library containing the called API and the name of the API function in idealized form (without parameters and return information). Let us consider the following trace ω as stated earlier:

```
kernel32.GetUserDefaultLCID,
kernel32.GetSystemInfo,
kernel32.RemoveDirectoryA,
kernel32.GetEnvironmentStringsA,
kernel32.LoadLibraryA,
kernel32.LoadLibraryA
```

where "kernel32" is a Windows system library and "GetUserDefaultLCID", "GetSystemInfo", "RemoveDirectoryA", "GetEnvironmentStringsA" and "LoadLibraryA" are APIs defined in kernel32 library. The contextual trace for **kernel32.RemoveDirectoryA** token, with components pre and $post$ of size 2 will be:

```
(kernel32.GetUserDefaultLCID,
kernel32.GetSystemInfo;
kernel32.RemoveDirectoryA-1;
kernel32.GetEnvironmentStringsA,
kernel32.LoadLibraryA)
```

For the central token of the context(x_i), we added a count which is the number of occurrences of the same token consecutively. This has the role of compacting context while keeping the same contextual trace expressive power.

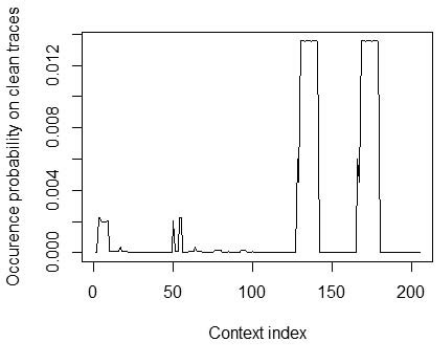

Figure 1. Occurrence probability of contexts from a trace corresponding to a malicious file in the multiset of clean contexts

In Figure 1 is plotted the occurrence probability for contexts from a malicious trace in the multiset of clean contexts. The analyzed sample was randomly selected from the collection of malicious samples. It can be noticed that there are several zones with a very low probability. Of all of these zones, we consider only those with a probability smaller than the threshold, *th*. In this case, these zones contain a sequence of contextual traces and, consequently, a sequence of tokens that is very likely to describe a malicious behavior. Also, these sequences can be considered a malicious property. The zones with a high probability correspond to the sequence of tokens that describe code pieces or actions which are commonly encountered in benign applications.

Figure 2 presents the same statistical analysis as earlier but performed on a clean trace. Even in this case we have several zones with low probability of occurrence. However, the probabilities are greater than the threshold and we can notice that the high probability zones are more frequent.

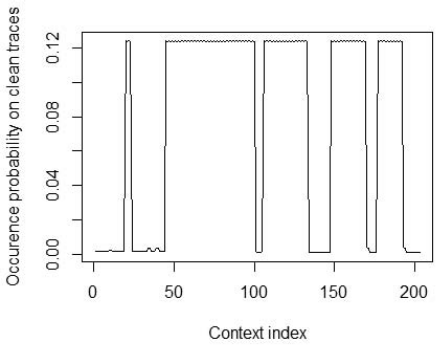

Figure 2. Occurrence probability of contexts from a trace corresponding to a benign file in the multiset of clean contexts

2) Malicious program filtering: The filtering method has as input a program and, consequently, a trace for the program. Also, it requires the databases containing analysis of contextual traces from clean instances and the databases of labeled zones, identified in malicious samples using algorithm described above. Firstly, the suspicious zones for the input trace are searched. If no zone is identified then the file is considered $CLEAN$ (similar with the clean set of samples). In case any suspicious zones are identified we have two verdicts:

- $MALICIOUS$ - the file is considered to be malicious if the identified sequence of labeled zones match a record from the database of known malicious zones. The number of malicious samples that have the same sequence of zones will be used as a score for setting priorities.
- $SUSPICIOUS$ - the identified sequence of zones does not match a malicious one, but since these zone have a small probability to occur on clean samples or they do not occur at all, it is considered that the file has suspicious content. In this case the number of identified suspicious zones will be used as score for setting priorities.

This filtering method (Figure 3) is intended to work as a preprocessing step in automatic analysis or detection systems. It has the purpose of identifying those samples which already contain known malicious code and of setting priorities for the suspicious ones. We can also map these labels to different families of malware and group the incoming samples into family based clusters.

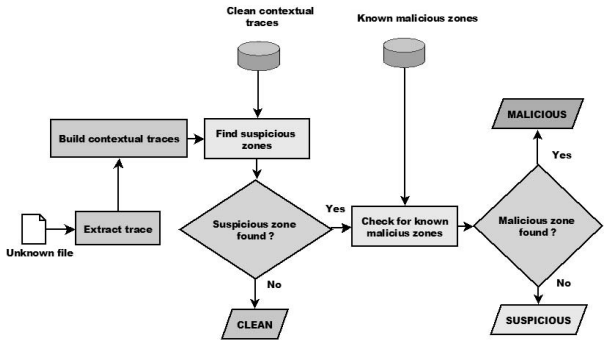

Figure 3. Filtering method overview

IV. Results

The training phase was performed using samples provided by Bitdefender Laboratories. The clean samples collection contains operating system files and more or less popular programs, collected over a period of 3 years. The number of collected samples was 3‘382‘650. The malicious samples were collected over a period of 3 months (February, March and April) of 2015 with a total number of 105‘458. On

the testing phase, we used a number of 25‘862 malicious samples collected in May 2015.

We let the length of pre and $post$ sequences to be 2, which means that a contextual trace will contain at least 5 tokens. The value has been chosen based on internal study that an action or a behavior can be described on average by 5 APIs. Also, this choice was made considering the practical limits of the dynamic analysis. The traces extracted have to describe a behavior. Therefore, the contexts could only be extracted from traces with length greater or equal to 5. For the training data collection, 2‘025‘897 clean records and 75‘516 malicious records passed this condition. For the test collection, 18‘839 records passed the already mentioned condition.

The unique contextual traces built from traces based on clean files were 1‘963‘561, with a number of total occurrences in the multiset $CtxB$ of 531‘489‘706.

We used these contextual traces to extract the sequences of zones. From the malicious traces, we identified a number of 6‘707 unique sequences, found on 44‘924 malicious traces. These sequences were labeled and stored as signatures.

With these signatures, the filtering method has classified as malicious 4‘107 samples and 10‘790 as suspicious from the test collection. The filtering rates are shown in Table II.

Verdict	Number of samples	Percentage
Malicious	4‘107	22%
Suspicious	10‘790	57%
Clean	3‘942	21%
Total	18‘839	100%

Table II
FILTERING RATES

From the test collection we randomly selected 180 files (1%) which were verified by human analysis. Only 3 false positives (file flagged as malicious or suspicious but after analysis was considered to be benign). We also found 30 files that were flagged as clean but were malicious(false negatives). Considering the false positive rate, this method can be considered reliable. Due to the detection rate of almost 80%, this method can be used for filtering purposes as a complementary solution.

Further we will show how the use of this method can reduce the response time of an automated analysis system on detecting a new unknown malware.

Let us consider that an automated analysis system requires a time t for analysis and signature generation for a new unknown malware. We randomly selected 5 new samples from an incoming collection of our automated system and set an initial priority for them (Table III).

Then we applied the filtering method using clean contexts information and extracted zones. Based on verdict and the score we changed priority for the analysis system. The results are presented in Table IV.

File ID.	Priority	Response time
00899	1	t
10598	2	$2 \times t$
30853	3	$3 \times t$
61912	4	$4 \times t$
96921	5	$5 \times t$

Table III
PRIORITY FOR AUTOMATED ANALYSIS SYSTEM WITHOUT FILTERING

File ID.	Priority	Delta response time	Verdict and score
00899	5	-4	clean
10598	3	-1	suspicious (19 zones)
30853	2	$+1$	malicious (26 similar)
61912	4	0	suspicious (2 zones)
96921	1	$+4$	malicious (1904 similar)

Table IV
PRIORITY FOR AUTOMATED ANALYSIS SYSTEM AFTER FILTERING

After the files were processed by the automated system, the file with ID 00899 was considered clean and the rest of the files were malicious. After a human validation, we confirmed that all five were indeed correctly classified. After studying prevalence[5] of these malicious files, we observed that the file with ID 96921 had the highest prevalence among them. Therefore, reducing the delay for analyzing a prevalent sample by filtering and adjusting priority has a great impact on end-user protection and usability.

We performed another experiment to see how much our proposed filtering method will increase the response time on large inputs. The experiment was done using for test data the combined collections of clean and malicious traces. The total size of collection is 2‘101‘413. In this case we know that all malicious traces and, implicitly, the corresponding files will be identified by our filtering method. A file is identified by the MD5 hash generated on its content. We considered the initial priority the lexicographic order of the file’s identification. In this case, the last malicious file will have the index 2‘101‘366 in queue for analysis system. After the filtering process, in worst case the file will have the index 75‘516 for analysis system’s queue. If we presume that for a single file the analysis system requires about 30 seconds, the response time for the usual lexicographic order for the last malicious file will be almost 2 years. After using the filtering technique the response time decreases to 26 days. These are extreme cases which use the assumption that filtering has 100% detection rate and no parallel computation is performed. As seen on test data, the detection rate of identified malicious and suspicious files is around 79%, so the filtering method applied on a usual input of 30.000 files

[5]spreading rate among users

per day, will improve the response time for the prevalent ones.

V. Conclusion

Due to the fact that malware nowadays are designed using a polymorphic engine and are targeting at least one security product, having a method that can quickly identify this type of threats is a necessity. In this paper we have presented a method that proved to be successful in optimizing and improving both the detection rate and the detection update time for large data sets of malicious files. This method also proved to be highly effective in differentiating between malware families.

We plan to extend this method for other forms of malicious content (other than binaries), such as scripting languages, emails, exploit payloads, network traffic, etc. In case of binaries, we will also develop library identification methods based on these techniques. These methods will help us construct our traces on code that is particular for that application and not some code specific to a certain library that might be a common thing between different programs that are using that library.

References

[1] Manoun Alazab, Robert Layton, Sitalakshmi Venkatraman, and Paul Watters. Malware detection based on structural and behavioural features of api calls. In *Proceedings of the 1st International Cyber Resilience Conference*, pages 1–10. Security Research Centre, Edith Cowan Univ., 2010.

[2] Philippe Beaucamps, Isabelle Gnaedig, and Jean-Yves Marion. Abstraction-based malware analysis using rewriting and model checking. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, volume 7459 of *Lecture Notes in Computer Science*, pages 806–823. Springer, 2012.

[3] Michael Bloem, Tansu Alpcan, and Tamer Basar. A robust control framework for malware filtering. *CoRR*, abs/0911.2293, 2009.

[4] Michael Bloem, Tansu Alpcan, Stephan Schmidt, and Tamer Basar. Malware filtering for network security using weighted optimality measures. In *Proceedings of the IEEE International Conference on Control Applications, CCA 2007, Singapore, October 1-3, 2007*, pages 295–300. IEEE, 2007.

[5] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In Gautam Shroff, Pankaj Jalote, and Sriram K. Rajamani, editors, *Proceeding of the 1st Annual India Software Engineering Conference, ISEC 2008, Hyderabad, India, February 19-22, 2008*, pages 5–14. ACM, 2008.

[6] Nilesh Prajapati Kevadia Kaushal, Prashant Swadas. Metamorphic malware detection using statistical analysis. *International Journal of Soft Computing and Engineering*, 2(3):49–53, 2012.

[7] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In Klaus Julisch and Christopher Krügel, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment, Second International Conference, DIMVA 2005, Vienna, Austria, July 7-8, 2005, Proceedings*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2005.

[8] Felix Leder, Bastian Steinbock, and Peter Martini. Classification and detection of metamorphic malware using value set analysis. In *4th International Conference on Malicious and Unwanted Software, MALWARE 2009, Montréal, Quebec, Canada, October 13-14, 2009*, pages 39–46. IEEE, 2009.

[9] Hugo Daniel Macedo and Tayssir Touili. Mining malware specifications through static reachability analysis. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *Lecture Notes in Computer Science*, pages 517–535. Springer, 2013.

[10] Shuhui Qi, Ming Xu, and Ning Zheng. A malware variant detection method based on byte randomness test. *JCP*, 8(10):2469–2477, 2013.

[11] R. Rivest. The MD5 Message-Digest Algorithm, 1992. RFC 1321.

[12] Igor Santos, Yoseba K. Penya, Jaime Devesa, and Pablo Garcia Bringas. N-grams-based file signatures for malware detection. In José Cordeiro and Joaquim Filipe, editors, *ICEIS 2009 - Proceedings of the 11th International Conference on Enterprise Information Systems, Volume AIDSS, Milan, Italy, May 6-10, 2009*, pages 317–320, 2009.

[13] Fu Song and Tayssir Touili. Pushdown model checking for malware detection. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2012.

[14] S. Momina Tabish, M. Zubair Shafiq, and Muddassar Farooq. Malware detection using statistical analysis of byte-level file content. In Hsinchun Chen, Marc Dacier, Marie-Francine Moens, Gerhard Paass, and Christopher C. Yang, editors, *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics, Paris, France, June 28, 2009*, pages 23–31. ACM, 2009.

[15] Wing Wong and Mark Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, 2006.